

# **Procura de padrões em sequências**

# Objetivo

Dada uma sequência num dado alfabeto, procurar **todas as ocorrências** de um padrão nessa sequência

Vamos assumir que o padrão é fixo.

# Algoritmo “naive”: procura em todas as posições

P: word

S: There would have been a time for such a word

# comparações

S: **T**here would have been a time for such a word  
word

1

S: **T**here would have been a time for such a word  
-word

2

S: There **w**ould have been a time for such a word  
-----word

9

S: There would have been a time for such a **w**ord  
-----word

46

## Algoritmo “naive”: procura em todas as posições

```
seq = input("Sequence:")
pattern = input("Pattern:")

res = []

for i in range(len(seq)-len(pattern)+1):           # loop over alignments
    j = 0
    while j < len(pattern) and pattern[j] == seq[i+j]: # loop over characters and compare
        j = j + 1

    if j == len(pattern):                          # If match append the alignment position
        res.append(i)

print("Pattern occurs in positions:", res)
```

# Complexidade do algoritmo naive

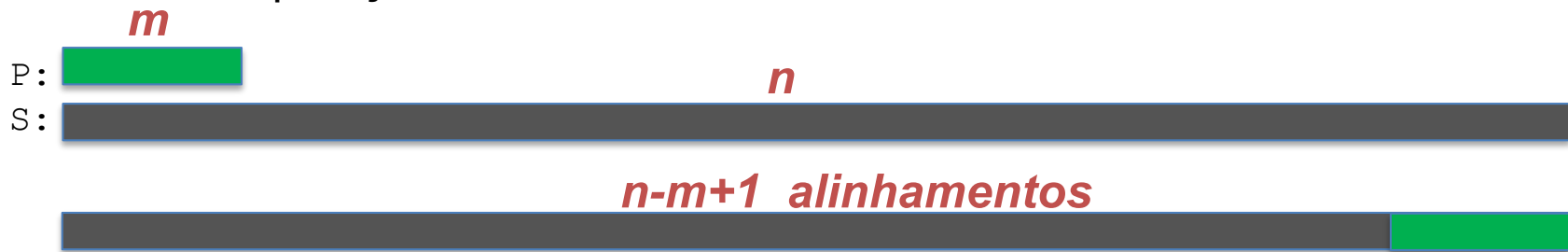
No seu pior caso o algoritmo naive anterior tem complexidade:

$$O(m(n-m+1)) \approx O(mn)$$

$m$  – comprimento do padrão

$n$  – comprimento da sequência

Percorrem-se todas as posições possíveis ( $n-m+1$ ) e, em cada caso fazem-se no pior caso  $m$  comparações.



Como melhorar a eficiência destes algoritmos ?

# Pré-processamento do padrão

Uma das alternativas para melhorar o desempenho destes algoritmos é realizar o **pré-processamento do padrão** de uma forma inteligente.


Objectivo:

- Realizar a procura de vários padrões com uma **única passagem** pela sequência, e/ou
- Optimizar a procura do(s) **mesmo(s) padrão(ões) em várias sequências** distintas

Iremos abordar 3 alternativas:

- Algoritmos heurísticos
- **Autómatos finitos**
- Árvore com os padrões **(tries) – próxima aula !!**

# Algoritmo de Boyer-Moore

S: There would have been a time for such a word  
-----word  
          

O carater **u** não ocorre no padrão P, assim podemos saltar dois alinhamentos.

S: There would have been a time for such a word  
-----word

Aprende com as comparações para saltar alinhamentos desnecessários.

# Algoritmo de Boyer-Moore

O algoritmo de Boyer-Moore é um algoritmo **heurístico** que permite tornar mais eficiente a procura de padrões em sequências

- o pior caso tem complexidade igual ao algoritmo naive;
- na maior parte dos casos permite ganhos significativos.

Algoritmo baseia-se num pré-processamento do padrão segundo duas regras; estas permitem, quando há uma falha no match entre o padrão e a sequência avançar o máximo de posições possível.

Será implementado na classe *BoyerMoore* (material suplementar à aula).

Boyer, RS and Moore, JS. "A fast string searching algorithm."  
Communications of the ACM 20.10 (1977): 762-772.



# Algoritmo de Boyer Moore - descrição

Testa os alinhamentos da esquerda para a direita e as comparações de caracteres da direita para a esquerda:

```
S: There would have been a time for such a word  
-----word
```


Quando há uma falha no match entre o padrão e a sequência, existem duas regras que permitem avançar a pesquisa:

- **Bad-character rule**
- **Good suffix rule**

# Boyer Moore: bad character rule


Avançar para a próxima ocorrência no padrão do símbolo que falhou (ou se não existir avançar o máximo possível).

S: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A  
P: C C T T T T G C



Avança para a próxima ocorrência do símbolo que falhou (C).

S: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A  
P:   C C T T T T G C



O símbolo que falhou (A) não existe no padrão logo avança o máximo possível.

S: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A  
P: C C T T T T G C

# Boyer Moore: good suffix rule

Avançar para a próxima ocorrência no padrão da parte que fez match antes de falhar. Se o sufixo não ocorre de novo, pode avançar tamanho do padrão.

S: C G T G C C T A C T T A C T T A C T T A C G C G A A  
P: C T T A C T T A C

Avança para a próxima ocorrência do sufixo.

S: C G T G C C T A C T T A C T T A C T T A C G C G A A  
P: C T T A C T T A C

Sufixo não ocorre mas parte do prefixo do padrão faz match com um sufixo do match anterior (no exemplo "C")

S: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A  
P: C T T A C T T A C

## Boyer Moore: ambas as regras

O algoritmo aplica as duas regras e avança **o máximo avanço** dado por cada uma delas.

S: G T T A T A G C **T** G A T C G C G G C G T A G C G G C G A A **bc: 7**

P: G **T** A G C G G C **G** **gs: 1**

S: G T T A T A G C T G A T **C** **G C G** G C G T A G C G G C G A A **bc: 1**

P: G T A **G C** **G** **G C G**

S: G T T A T A G C T G A T **C** G C G G C G T A G C G G C G A A **bc: 3**

P: **G** T **A** G C G G C G **gs: 8**

S: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A  
P: G T A G C G G C G

# Boyer Moore: pré-processamento

G T A G C G G C G  
0 1 2 3 4 5 6 7 8

Tamanho do padrão = 9

## Bad Character rule:

Σ	A	C	T	G
occ	2	7	1	8

Última posição onde o caractere ocorre no padrão.

Se não ocorrer: -1.

Avança: **Tamanho do padrão - occ**

## Good Suffix rule:

P	-	G	T	A	G	C	G	G	C	G
s	8	8	8	8	8	8	3	8	2	1

O processamento envolve os 2 casos, sufixo ocorre de novo no padrão, ou um sufixo do match é parte do prefixo do padrão. (ver algoritmo)

# Exercício

- Implementar algoritmo de Boyer-Moore numa classe
- Métodos:
  - Construtor (a partir do alfabeto e de um padrão)
  - Métodos que fazem o pré-processamento BCR e GSR (chamados no construtor)
  - Método que identifica as posições de match do padrão numa sequência (dada como argumento)

## Exercício – processamento BCR (pseudo-código)

Criar dicionário vazio *self.occ*

PARA cada símbolo *s* no alfabeto

    Criar entrada no dicionário *self.occ* com chave *s* e valor -1

PARA cada índice *j* entre 0 e o tamanho do padrão

    Procurar entrada no dicionário *self.occ* com chave igual ao caractere do padrão na posição *j* e atualizar o valor para *j*

```
def process_bcr (self):
```

```
    ...
```

```
Padrão – self.pattern
```

```
Alfabeto – self.alphabet
```

## Exercício – processamento GSR (pseudo-código)

Criar listas *self.f* e *self.s* de tamanho igual ao do padrão + 1, cada uma inicializada com zeros  
Inicializar o valor de *i* com o tamanho do padrão e o de *j* com o tamanho do padrão + 1

*self.f[i] <- j*

ENQUANTO *i* > 0

ENQUANTO *j* <= tamanho do padrão E *self.pattern[i-1] != self.pattern[j-1]*:

SE *self.s[j] = 0*: *self.s[j] <- j - i*

*j <- self.f[j]*

*i <- i - 1; j <- j - 1*

*self.f[i] <- j*

*j <- self.f[0]*

PARA *i* entre 0 e o tamanho do padrão

SE *self.s[i] = 0*: *self.s[i] <- j*

SE *i = j*: *j <- self.f[j]*

*def process\_gsr (self):*

...



# Exercício – procurar padrão (pseudo-código)

Criar *res* como lista vazia

*i* <- 0

ENQUANTO *i* <= tamanho da sequência *text* – tamanho do padrão

*j* <- tamanho do padrão – 1

    ENQUANTO *j* >= 0 E *self*.pattern[*j*] = *text*[*j*+*i*]

*j* <- *j* – 1

    SE *j* < 0:

        Adicionar *i* no final da lista *res*

*i* <- *i* + *self*.s[0]

    SENÃO:

*c* <- *text*[*j*+*i*]

        Somar a *i* o maior valor entre *self*.s[*j*+1] e *j*-*self*.occ[*c*]

RETORNAR *res*

def *search\_pattern* (*self*, *text*):

...

# Autómatos finitos

**Autómatos finitos (AF)** podem ser descritos como “máquinas” que processam uma sequência de símbolos da esquerda para a direita.

Os AFs alteram o seu estado interno à medida que processam os caracteres; novo estado depende apenas do anterior e do último caractere lido.

Escolhendo adequadamente as transições pode determinar-se se o padrão está contido até à posição lida.

Um AF criado a partir de um padrão  $p$  pode determinar todas as posições de  $p$  numa sequência com uma única passagem.

# Autómatos finitos

Definição: um **autômato** é uma estrutura

$$M = (Q, A, q_0, \delta, F)$$

**Q** – conjunto de estados

**A** – alfabeto (conjunto de possíveis símbolos)

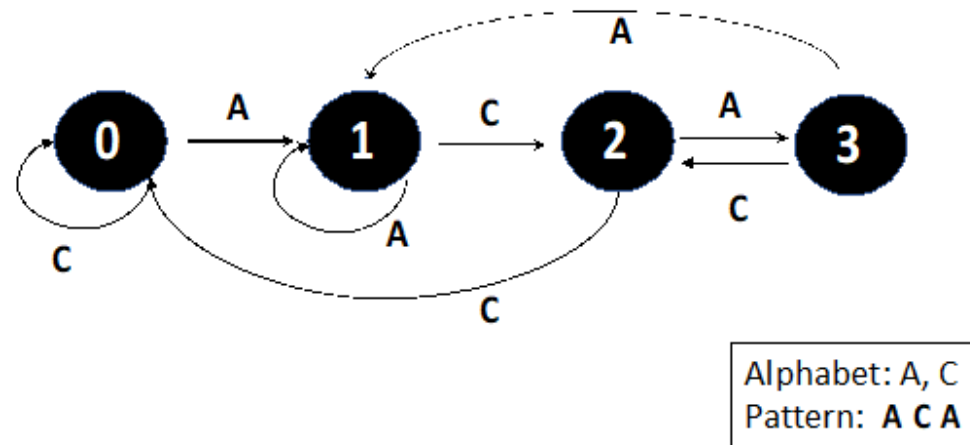
**$q_0$**   $\in Q$  – estado inicial

**$\delta$** :  $Q, A \rightarrow Q$  - função de transição

**F** (sub-conjunto de  $Q$ ) – conjunto de estados finais

# Exemplo de um AF para um padrão

State	Symbol	Next state
0	A	1
0	C	0
1	A	1
1	C	2
2	A	3
2	C	0
3	A	1
3	C	2



Sequence		C	A	C	A	A	C	A	A
State	0	0	1	2	3	1	2	3	1
Occurrence					1			4	

# Construindo um AF para um padrão

Dado um padrão  $p$  de tamanho  $m$  definido num alfabeto  $A_p$ , um autômato capaz de detectar  $p$  em qualquer sequência tem a seguinte estrutura:

- $Q = \{ 0, \dots, m \}$
- $A = A_p$
- $q_0 = 0$
- $F = \{ m \}$
- $\delta(q, a) = \text{max\_overlap}(p_0 \dots p_{q-1}a, p)$

# Construindo um AF para um padrão

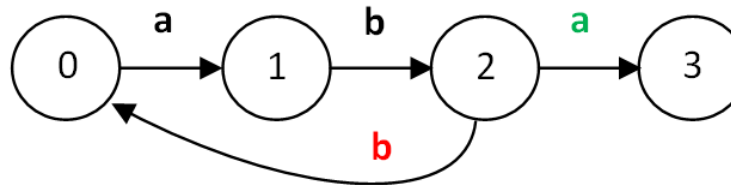
$\text{max\_overlap}(s1, s2)$ : função que dá o comprimento do *overlap* máximo entre  $s1$  e  $s2$ , ou seja, o maior sufixo de  $s1$  que é prefixo de  $s2$ :

Exemplo:  $A_p = \{a, b\}$   $p = \text{"ababab"}$

$\text{max\_overlap}(\text{"aba"}, \text{"ababab"}) = 3$

$\text{max\_overlap}(\text{"abb"}, \text{"ababab"}) = 0$

State	Symbol	Next state
2	a	3
2	b	0



## Procura de padrões com um AF

Tendo um autômato construído, este pode ser usado para eficientemente procurar ocorrências do padrão usado para o construir

A sequência é percorrida de forma linear (uma única vez), sendo o estado atual do padrão atualizado de acordo com a tabela de transições dado o símbolo na sequência (e o estado anterior)

Sempre que o estado atual é igual a ***m***, o padrão foi encontrado

# Complexidade dos AFs

Quanto à complexidade:

A tarefa de construir o AF pode ser realizada por algoritmos  $O(|A|.m)$

A tarefa de percorrer a sequência é  $O(n)$

No total teremos:  $O(n + |A|.m)$

O uso de AFs é especialmente eficiente quando queremos procurar o mesmo padrão num conjunto de sequências – uma tarefa comum em Bioinformática



# Exercício

- Implementar AFs definindo uma classe Automata
- Classe deverá ter variáveis para guardar o alfabeto, o número de estados, a tabela de transições
- Métodos:
  - Construtor (a partir do alfabeto e de um padrão)
  - Método que constrói a tabela de transições a partir do padrão (usado no construtor)
  - Método que aplica o AF a uma sequência (dada como entrada), dando como resultado a lista de estados
  - Método que identifica as posições de match do padrão representado pelo AF numa sequência (dada como entrada)